Piggybacking on Social Networks*

Aristides Gionis
Aalto University and HIIT
Espoo, Finland
aristides.gionis@aalto.fi

Flavio Junqueira
Microsoft Research
Cambridge, UK
fpj@microsoft.com

Vincent Leroy
Univ. of Grenoble – CNRS
Grenoble, France
vincent.leroy@imag.fr

Marco Serafini
QCRI
Doha, Qatar
mserafini@qf.org.qa

Ingmar Weber
QCRI
Doha, Qatar
ingmarweber@acm.org

ABSTRACT

The popularity of social-networking sites has increased rapidly over the last decade. A basic functionalities of social-networking sites is to present users with streams of events shared by their friends. At a systems level, materialized per-user views are a common way to assemble and deliver such event streams on-line and with low latency. Access to the data stores, which keep the user views, is a major bottleneck of social-networking systems. We propose to improve the throughput of these systems by using *social piggybacking*, which consists of processing the requests of two friends by querying and updating the view of a third common friend. By using one such *hub* view, the system can serve requests of the first friend without querying or updating the view of the second. We show that, given a social graph, social piggybacking can minimize the overall number of requests, but computing the optimal set of hubs is an $\mathbf{NP}$-hard problem. We propose an $\mathcal{O}(\log n)$ approximation algorithm and a heuristic to solve the problem, and evaluate them using the full Twitter and Flickr social graphs, which have up to billions of edges. Compared to existing approaches, using social piggybacking results in similar throughput in systems with few servers, but enables substantial throughput improvements as the size of the system grows, reaching up to a 2-factor increase. We also evaluate our algorithms on a real social networking system prototype and we show that the actual increase in throughput corresponds nicely to the gain anticipated by our cost function.

1. INTRODUCTION

Social networking sites have become highly popular in the past few years. An increasing number of people use social networking applications as a primary medium of finding new and interesting information. Some of the most popular social networking applications include services like Facebook, Twitter, Tumblr or Yahoo! News Activity. In these applications, users establish connections with other users and share *events*: short text messages, URLs, photos, news stories, videos, and so on. Users can browse *event streams*, real-time lists of recent events shared by their contacts, on most social networking sites. A key peculiarity of social networking applications compared to traditional Web sites is that the process of information dissemination is taking place in a many-to-many fashion instead of the traditional few-to-many paradigm, posing new system scalability challenges.

In this paper, we study the problem of assembling event streams, which is the predominant workload of many social networking applications, e.g., 70% of the page views of Tumblr.[1] Assembling of event streams needs to be on-line, to include the latest events for every user, and very fast, as users expect the resulting event streams to load in fractions of a second.

To put our work in context and to motivate our problem definition, we describe the typical architecture of social networking systems, and we discuss the process of assembling event streams. We consider a system similar to the one depicted in Figure 1. In such a system, information about users, the social graph, and events shared by users are stored in *back-end data stores*. Users send requests, such as sharing new events or receiving updates on their event stream, to the social networking system through their browsers or mobile apps.

A large social network with a very large number of active users generates a massive workload. To handle this query workload and optimize performance, the system uses *materialized views*. Views are typically formed on a per-user basis, since each user sees a different event stream. Views can contain events from a user's contacts and from the user itself. Our discussion is independent of the implementation of the data stores; they could be relational databases, key-value stores, or other data stores.

The throughput of the system is proportional to the data transferred to and from the data stores; therefore, increasing the data-store throughput is a key problem in social networking systems.[2] In this paper, we propose optimization algorithms to reduce the load induced on data stores—the thick red arrows in Figure 1. Our algorithms make it possible to run the application using fewer data-store servers or, equivalently, to increase throughput with the same number of data-store servers.

Commercial social networking systems already use strategies to send fewer requests to the data-store servers. A system can group the views of the contacts of a user in two user-specific sets: the *push set*, containing contact views that are updated by the data-

*Work conducted while the authors were with Yahoo! Research.

[1] http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html

[2] http://www.facebook.com/note.php?note_id=39391378919

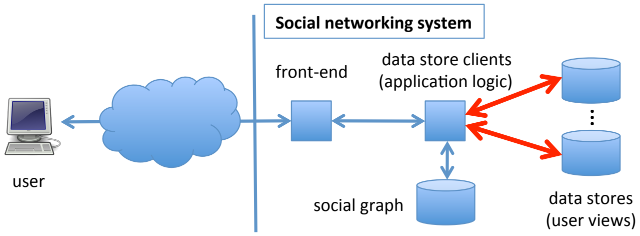


Figure 1: Simplified request flow for handling event streams in a social networking system. We focus on reducing the throughput cost of the most complex step: querying and updating data stores (shown with thick red arrows).

store clients when the user shares a new event, and the *pull set*, containing contact views that are queried to assemble the user's event stream. The collection of push and pull sets for each user of the system is called *request schedule*, and it has strong impact on performance. Two standard request schedules are *push-all* and *pull-all*. In push-all schedules, the push set contains all of user's contacts, while the pull set contains only the user's own view. This schedule is efficient in read-dominated workloads because each query generates only one request. Pull-all schedules are specular, and are better suited for write-dominated workloads. More efficient schedules can be identified by using a hybrid approach between pull- and push-all, as proposed by Silberstein et al. [11]: for each pair of contacts, choose between push and pull depending on how frequently the two contacts share events and request event streams. This approach has been adopted, for example, by Tumblr.

In this paper we propose strictly cheaper schedules based on *social piggybacking*: the main idea is to process the requests of two contacts by querying and updating the view of a third common contact. Consider the example shown in Figure 2. For generality, we model a social graph as a directed graph where a user may follow another user, but the follow relationship is not necessarily symmetric. In the example, Charlie's view is in Art's push set, so clients insert every new event by Art into Charlie's view. Consider now that Billie follows both Art and Charlie. When Billie requests an event stream, social piggybacking lets clients serving this request pull Art's updates from Charlie's view, and so Charlie's view acts as a *hub*. Our main observation is that the high clustering coefficient of social networks implies the presence of many hubs, making hub-based schedules very efficient [10].

Social piggybacking generates fewer data-store requests than approaches based on push-all, pull-all, or hybrid schedules. With a push-all schedule, the system pushes new events by Art to Billie's view—the dashed thick red arrow in Figure 2(b). With a pull-all schedule, the system queries events from Art's view whenever Billie requests a new event stream—the dashed double green arrow in Figure 2(b). With a hybrid schedule, the system executes the cheaper of these two operations. With social piggybacking, the system does not execute any of them.

Using hubs in existing social networking architectures is very simple: it just requires a careful configuration of push and pull sets. In this paper, we tackle the problem of calculating this configuration, or in other words, the request schedule. The objective is to minimize the overall rate of requests sent to views. We call this problem the *social-dissemination* problem.

Our contribution is a comprehensive study of the problem of social-dissemination. We first show that optimal solutions of the social-dissemination problem either use hubs (as Charlie in Figure 2) or, when efficient hubs are not available, make pairs of users exchange events by sending requests to their view directly. This result reduces significantly the space of solutions that need to be explored, simplifying the analysis.

We show that computing optimal request schedules using hubs is $\mathbf{NP}$-hard, and we propose an approximation algorithm, which we call CHITCHAT. The hardness of our problem comes from the *set-cover problem*, and naturally, our approximation algorithm is based on a greedy strategy and achieves an $\mathcal{O}(\log n)$ guarantee. Applying the greedy strategy, however, is non-trivial, as the iterative step of selecting the most cost-effective subset is itself an interesting optimization problem, which we solve by mapping it to the *weighted densest-subgraph* problem.

We then develop a heuristic, named PARALLELNOSY, which can be used for very large social networks. PARALLELNOSY does not have the approximation guarantee of CHITCHAT, but it is a parallel algorithm that can be implemented as a MapReduce job and thus scales to real-size social graphs.

CHITCHAT and PARALLELNOSY assume that the graph is static; however, using a simple incremental technique, request schedules can be efficiently adapted when the social graph is modified. We show that even if the social graph is dynamic, executing an initial optimization pays off even after adding a large number of edges to the graph, so it is not necessary to optimize the schedule frequently.

Evaluation on the full Twitter and Flickr graphs, which have billions of edges, shows that PARALLELNOSY schedules can improve *predicted throughput* by a factor of up to 2 compared to the state-of-the-art scheduling approach of Silberstein et al. [11].

Using a social networking system prototype, we show that the *actual throughput* improvement using PARALLELNOSY schedules compared to hybrid scheduling is significant and matches very well our predicted improvement. In small systems with few servers the throughput is similar, but the throughput improvement grows with the size of the system, becoming particularly significant for large social networking systems that use hundreds of servers to serve millions, or even billions, of requests.[3] With 500 servers, PARALLELNOSY increases the throughput of the prototype by about 20%; with 1000 servers, the increase is about 35%; eventually, as the number of server grows, the improvement approaches the predicted 2-factor increase previously discussed. In absolute terms, this may mean processing millions of additional requests per second.

We also compare the performance of CHITCHAT and PARALLELNOSY on large samples of the actual Twitter and Flickr graphs. CHITCHAT significantly outperforms PARALLELNOSY, showing that there is potential for further improvements by making more complex social piggybacking algorithms scalable.

Overall, we make the following contributions:

- Introducing the concept of social piggybacking, formalizing the social dissemination problem, and showing its $\mathbf{NP}$-hardness;
- Presenting the CHITCHAT approximation algorithm and showing its $\mathcal{O}(\log n)$ approximation bound;
- Presenting the PARALLELNOSY heuristic, which can be parallelized and scaled to very large graphs;
- Evaluating the predicted throughput of PARALLELNOSY schedules on full Twitter and Flickr graphs;
- Measuring actual throughput on a social networking system prototype;
- Comparing CHITCHAT and PARALLELNOSY on samples of the Twitter and Flickr graphs to explore possible further gains.

[3] For an example, see: http://gigaom.com/2011/04/07/facebook-this-is-what-webscale-looks-like/

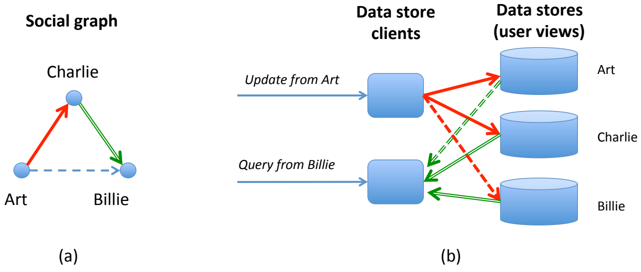


Figure 2: Example of social piggybacking. Pushes are thick red arrows, pulls double green ones. (a) The edge from Art to Billie can be served through Charlie if Art pushes to Charlie and Billie pulls from Charlie. (b) Charlie's view is a hub. Existing approaches unnecessarily issue one of the dashed requests.

Roadmap. In Section 2 we discuss our model and present a formal statement of the problem we consider. In Section 3 we present our algorithms, which we evaluate in Section 4. We discuss the related work in Section 5, and Section 6 concludes the work.

2. SOCIAL DISSEMINATION PROBLEM

We formalize the social-dissemination problem as a problem of propagating events on a social graph. The goal is to efficiently broadcast information from a user to its neighbors. Dissemination must satisfy bounded staleness, a property modeling the requirement that event streams shall show events almost in real time. We then show that the only request schedules satisfying bounded staleness let each pair of users communicate either using direct push, or direct pull, or social piggybacking. Finally, we analyze the complexity of the social-dissemination problem and show that our results extend to more complex system models with active stores.

2.1 System model

We model the social graph as a directed graph $G = (V, E)$. The presence of an edge $u \rightarrow v$ in the social graph indicates that the user v *subscribes* to the events produced by u. We will call u a *producer* and v a *consumer*. Symmetric social relationships can be modeled with two directed edges $u \rightarrow v$ and $v \rightarrow u$.

A user can issue two types of requests: sharing an *event*, such as a text message or a picture, and requesting an updated *event stream*, a real-time list of recent events shared by the producers of the user.

For the purpose of our analysis, we do not distinguish between nodes in the graph, the corresponding users, and their materialized views. There is one view per user. A user view contains events from the user itself and from the other users it subscribed to; sending events to uninterested users results in unnecessary additional throughput cost, which is the metric we want to minimize.

Definition 1 (View) *A view is a set of events such that if an event produced by user u is in the view of user v, then $u = v$ or $u \rightarrow v \in E$.*

Event streams and views consist of a finite list of events, filtered according to application-specific relevance criteria. Different filtering criteria can be easily adapted in our framework; however, for generality purposes, we do not explicitly consider filtering criteria but instead assume that all necessary past events are stored in views and returned by queries.

A fundamental requirement for any feasible solution is that event streams have *bounded staleness*: each event stream assembled for a user u must contain every recent event shared by any producers of u; the only events that are allowed to be missing are those shared at most Θ time units ago. The specific value of the parameter Θ may depend on various system parameters, such as the speed of networks, CPUs, and external-memories, but it may also be a function of the current load of the system. The underlying motivation of bounded staleness is that typical social applications must present near real-time event streams, but small delays may be acceptable.

Definition 2 (Bounded staleness) *There exists finite time bound Θ such that, for each edge $u \rightarrow v \in E$, any query action of v issued at any time t in any execution returns every event posted by u in the same execution at time $t - \Theta$ or before.*

Note that the staleness of event streams is different from request latency: a system might assemble event streams very quickly, but they might contain very old events. Our work addresses the problem of request latency indirectly: improving throughput makes it more likely to serve event streams with low latency.

In the system of Figure 2, the request schedule determines which edges of the social graph are included in the push and pull sets of any user. In our formal model, we consider two global pus$H$ and pul$L$ sets, called H and L respectively, both subsets of the set of edges E of the social graph. If a node u *pushes* events to a node v in the model, this corresponds, in an actual system like the one shown in Figure 2, to data-store clients updating the view of the user v with all new events shared by user u whenever u shares them. Similarly, if a node v *pulls* events from a node u, this corresponds to data-store clients sending a query request to the view of the user u whenever v requests its event stream. For simplicity, we assume that users always access their own view with updates and queries.

Definition 3 (Request schedule) *A* request schedule *is a pair (H, L) of sets, with a push set $H \subseteq E$ and a pull set $L \subseteq E$. If v is in the push set of u, we say that $u \rightarrow v \in H$. If u is in the pull set of v, we say that $u \rightarrow v \in L$.*

It is important to note that all existing push-all, pull-all, and hybrid schedules described in Section 1 are sub-classes of the request schedule class defined above.

The goal of social dissemination is to obtain a request schedule that minimizes the throughput cost induced by a workload on a social networking system. We characterize the throughput cost of a workload as the overall rate of query and updates it induces on data-store servers. The workload is characterized by the *production rate* $r_p(u)$ and the *consumption rate* $r_c(u)$ of each user u. These rates indicate the average frequency with which users share new events and request event streams, respectively. Given an edge $u \rightarrow v$, the cost incurred if $u \rightarrow v \in H$ is $r_p(u)$, because every time u shares a new event, an update is sent to the view of v; similarly, the cost incurred if $u \rightarrow v \in L$ is $r_c(v)$, because every event stream request from v generates a query to the view of u.

The cost of the request schedule (H, L) is thus:

$$c(H, L) = \sum_{u \rightarrow v \in H} r_p(u) + \sum_{u \rightarrow v \in L} r_c(v).$$

This expression does not explicitly consider differences in the cost of push and pull operations, modeling situations where the messages generated by updates and queries are very small and have similar cost. In order to model scenarios where the cost of a pull operation is k times the cost of a push, independent of the specific throughput metric we want to minimize (*e.g.*, number of messages, number of bytes transferred), it is sufficient to multiply all consumption rates by a factor k. Similarly, multiplying all production rates by a factor k models systems where a push is more expensive than a pull. Note that the cost of updating and querying a user's own view is not represented in the cost metric because it is implicit.

2.2 Problem definition

We now define the problem that we address in this paper.

Problem 1 (DISSEMINATION) *Given a graph $G = (V, E)$, and a workload with production and consumption rates $r_p(u)$ and $r_c(u)$ for each node $u \in V$, find a request schedule (H, L) that guarantees bounded staleness, while minimizing the cost $c(H, L)$.*

In this paper, we propose solving the DISSEMINATION problem using social piggybacking, that is, making two nodes communicate through a third common contact, called hub. Social piggybacking is formally defined as follows.

Definition 4 (Piggybacking) *An edge $u \to v$ of a graph $G(V, E)$ is covered by piggybacking through a hub $w \in V$ if there exists a node w such that $u \to w \in E$, $w \to v \in E$, $u \to w \in H$, and $w \to v \in L$.*

Let Δ be the upper bound on the time it takes for a system to serve a user request. Piggybacking guarantees bounded staleness with $\Theta = 2\Delta$. In fact, it turns out that admissible schedules transmit events over a social graph edge $u \to v$ only by pushing to v, pulling from u, or using social piggybacking over a hub.

Theorem 1 *Let (H, L) be a request schedule that guarantees bounded staleness on a social graph $G = (V, E)$. Then for each edge $u \to v \in E$, it holds that either (i) $u \to v \in H$, or (ii) $u \to v \in L$, or (iii) $u \to v$ is covered by piggybacking through a hub $w \in V$.*

PROOF. As we already discussed, all three operations satisfy the guarantee of bounded-time delivery. We will now argue that they are the only three such operations.

Assume that the edge $u \to v$ is not served directly, but via a path $p = u \to w_1 \to \ldots \to w_k \to v$. If the length of the path p is 2, i.e., if $k = 1$, then simple enumeration of all cases for paths of length 2 shows that social piggybacking is the only case that satisfies bounded staleness in each execution. For example, assume that both the edges $u \to w_1$ and $w_1 \to v$ are push edges. Then, delivery of an event requires that user w_1 will take some action within a certain time bound. However, since the user w_1 may remain idle for an arbitrarily long time, we cannot guarantee bounded staleness.

For longer paths a similar argument holds. In particular, for paths such that $k > 1$, the information has to propagate along some edge $w_i \to w_{i+1}$. The information cannot propagate along the edge $w_i \to w_{i+1}$ without one of the users w_i or w_{i+1} taking an action, and clearly we can assume that there exist executions in which both w_i or w_{i+1} remain idle after u has posted an event and before the next query of v. □

Even considering only the solution space restricted by Theorem 1, Problem 1 is $\mathbf{NP}$-hard. The proof, which uses a reduction from the SETCOVER problem, is omitted due to lack of space.

Theorem 2 *The* DISSEMINATION *problem is* $\mathbf{NP}$*-hard.*

So far we have considered systems where data-store servers react only to client operations. We can call data stores that only react to user request *passive stores*. Some data-store middleware enables data-store servers to propagate information among each other too. We generalize our result by considering a more general class of systems called *active stores*, where request schedules do not only include push and pull sets, but also *propagation sets* that are defined as follows:

Definition 5 (Propagation sets) *Each edge $w \to u$ is associated with a propagation set $P_u(w) \subseteq V$, which contains users who are common subscribers of u and w. If the view of u stores for the first time an event e produced by w, the data-store server pushes e to the view of every user $v \in P_u(w)$.*

We restrict the propagation of events to their subscribers to guarantee that a view only contains event from friends of the corresponding user. We only consider active policies where data stores take actions *synchronously*, when they receive requests. Some data stores can push events *asynchronously* and periodically: all updates received over the same period are accumulated and considered as a single update. Such schedules can be modeled as synchronous schedules having an upper bound on the production rates, determined based on the accumulation period and the communication latency between servers. Longer accumulation periods reduce throughput cost but also increase staleness, which can be problematic for highly interactive social networking applications.

The only difference between active and passive schedules is that the formers can determine chains of pushes $u \to w_1 \to \ldots \to w_k$. However, a chain of this form can be simulated in passive stores by adding each edge $u \to w_i$ to H, resulting in lower or equal latency and equal cost. This is formally shown by the following equivalence result. The proof is omitted for lack of space.

Theorem 3 *Any schedule of an active-propagation policy can be simulated by a schedule of a passive-propagation policy with no greater cost.*

This result implies that we do not need to consider active propagation in our analysis.

3. ALGORITHMS

This section introduces two algorithms to solve the DISSEMINATION problem. We have shown that the problem is $\mathbf{NP}$-hard, so we propose an approximation algorithm, called CHITCHAT, and a more scalable parallel heuristic, called PARALLELNOSY.

3.1 The CHITCHAT approximation algorithm

In this section we describe our approximation algorithm for the DISSEMINATION problem, which we name CHITCHAT. Not surprisingly, since the DISSEMINATION problem asks to find a schedule that *covers* all the edges in the network, our algorithm is based on the solution used for the SETCOVER problem.

For completeness we recall the SETCOVER problem: We are given a *ground set* T and a collection $\mathcal{C} = \{A_1, \ldots, A_m\}$ of subsets of T, called *candidates*, such that $\bigcup_i A_i = T$. Each set A in $\mathcal{C}$ is associated with a cost $c(A)$. The goal is to select a subcollection $\mathcal{S} \subseteq \mathcal{C}$ that covers all the elements in the ground set, i.e., $\bigcup_{A \in \mathcal{S}} A = T$, and the total cost $\sum_{A \in \mathcal{S}} c(A)$ of the sets in the collection $\mathcal{S}$ is minimized.

For the SETCOVER problem, the following simple *greedy* algorithm is folklore [5]: Initialize $\mathcal{S} = \emptyset$ to keep the iteratively growing solution, and $Z = T$ to keep the uncovered elements of T. Then as long as Z is not empty, select the set $A \in C$ that minimizes the cost per uncovered element $\frac{c(A)}{|A \cap Z|}$, add the set A to the

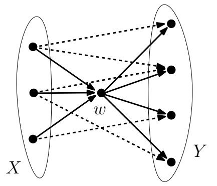


Figure 3: A hub-graph used in the mapping of DISSEMINATION **to** SETCOVER **problem. Solid edges must be served with a push (if they point to w) or a pull (if they point from w). Dashed edges are covered indirectly.**

solution ($\mathcal{S} \leftarrow \mathcal{S} \cup \{A\}$) and update the set of uncovered elements ($Z \leftarrow Z \setminus A$). It can be shown [5] that this greedy algorithm achieves a solution with approximation guarantee $\mathcal{O}(\log \Delta)$, where $\Delta = \max\{|A|\}$ is the size of the largest set in the collection $\mathcal{C}$. At the same time, this logarithmic guarantee is essentially the best one can hope for, since Feige showed that the problem is not approximable within $(1 - o(1)) \ln n$, unless $\mathbf{NP}$ has quasi-polynomial time algorithms [7].

The goal of our SETCOVER variant is to identify request schedules that optimize the DISSEMINATION problem. The ground set to be covered consists of all edges in the social graph. The solution space we identified in Section 2 indicates that the collection C contains two kinds of subsets: edges that are served directly, and edges that are served through a hub. Serving an edge $u \to v \in E$ directly through a push or a pull corresponds to covering using a singleton subset $\{u \to v\} \in C$. The algorithm chooses between push and pull according to the hybrid strategy of Silberstein et al. [11]. A hub like the one of Figure 2(a) is a subset that covers three edges using a push and a pull; the third edge is served indirectly. Every time the algorithm selects a candidate from C, it adds the required push and pull edges to the solution, the request schedule (H, L).

A straightforward application of the greedy algorithm described above has exponential time complexity. The iterative step of the algorithm must select a candidate from C, which has exponential cardinality because it contains all possible hubs. To our rescue comes a well-known property about applying the greedy algorithm for solving the SETCOVER problem: a sufficient condition for applying the greedy algorithm on SETCOVER is to have a *polynomial-time oracle* for selecting the set with the minimum cost-per-element. The oracle can be invoked at every iterative step in order to find an (approximate) solution of the SETCOVER problem without materializing all elements of C. This makes the cardinality of C irrelevant.

The algorithmic challenge of CHITCHAT is finding a polynomial time oracle for the DISSEMINATION problem. One key idea of CHITCHAT is to split the oracle problem in two sub-problems, both to be solved in polynomial time.

The first sub-problem is adding to C, for each node w, the *hub-graph* centered on w that covers the largest number of edges for the lowest cost. A hub-graph centered on w is a generalization of the sub-graph of Figure 2(a), as depicted in Figure 3. It is a sub-graph of the social graph where X is a set of nodes that w subscribes, and Y is a set of nodes that subscribe to w. We refer to such hub-graphs using the notation $G(X, w, Y)$.

The second sub-problem is selecting the best candidate of C. This is now simple since C contains a linear number of hub-graph elements and a quadratic number of singleton edges. If a hub-graph is selected, the edges from all nodes in X to w are set to be push, and the edges from w to all nodes in Y are set to be pull. All edges between nodes of X and Y are covered indirectly.

The first sub-problem, finding the hub-graph centered in a given node that covers most edges with lowest cost, is an interesting optimization problem in itself. In order to define the sub-problem, we associate to each node u of a hub-graph a weight $g(u)$ reflecting the cost of u. We set $g(x) = r_p(x)$ for all $x \in X$, that is, the cost of a push operation from x to w is associated to node x. Similarly we associate the weight $g(y) = r_c(y)$ for each $y \in Y$. For the hub node w, we set $g(w) = 0$. Let W and $E(W)$ be the set of nodes and edges of the hub-graph, respectively, and let $g(W) = \sum_{u \in W} g(u)$. The cost-per-element of the hub-graph is:

$$p(W) = \frac{g(W)}{|E(W)|}. \tag{1}$$

The sub-problem can thus be formulated as finding, for each node w of the social graph, the hub-graph $(W, E(W))$ centered on w that minimizes $p(W)$.

Careful inspection of Equation (1) motivates us to consider the following problem.

Problem 2 (DENSESTSUBGRAPH) *Let $G = (V, E)$ be a graph. For a set $S \subseteq V$, $E(S)$ denotes the set of edges of G between nodes of S. The* DENSESTSUBGRAPH *problem asks to find the subset S that maximizes the density function $d(S) = \frac{|E(S)|}{|S|}$.*

If we weight the nodes of S using the g function define above, we can obtain a *weighted* variant of this problem by replacing the density function $d(S)$ with $d_w(S) = |E(S)|/g(S)$.

Let G_w be the largest hub-graph centered in a node w, the one where X and Y include all producers and consumers of w, respectively. Any subgraph $(S, E(S))$ of G_w that maximizes $d_w(S)$ minimizes $p(S)$. Therefore, any solution of the weighted version of DENSESTSUBGRAPH will give us the hub-graph centered on w to be included in C.

Interestingly, although many variants of dense-subgraph problems are $\mathbf{NP}$-hard, Problem 2 can be solved exactly in polynomial time. Given that we are looking for a solution of the SETCOVER problem with a logarithmic approximation factor, we set for the simple greedy algorithm analyzed by Asahiro et al. [1] and later by Charikar [3]. This algorithm gives a 2-factor approximation for Problem 2, and its running time is linear in the number of edges in the graph. The algorithm is the following. Start with the whole graph. Until left with an empty graph, iteratively remove the node with the lowest degree (breaking ties arbitrarily) and all its incident edges. Among all subgraphs considered during the execution of the algorithm return the one with the maximum density.

The above algorithm works for the case that the density of a subgraph is $d(S)$. In our case we want to maximize the weighted-density function $d_w(S)$. Thus we modify the greedy algorithm of Asahiro et al. and Charikar as follows. In each iteration, instead of deleting the node with the lowest degree, we delete the node that minimizes a notion of *weighted degree*, defined as $d_g(u) = \frac{d(u)}{g(u)}$, where $d(u)$ is the normal notion of degree of node u. We can show that this modified algorithm yields a factor-2 approximation for the weighted version of the DENSESTSUBGRAPH problem.

Lemma 1 *Given a graph $G_w = (S, E(S))$, there exists a linear-time algorithm solving the weighted variant of the* DENSESTSUBGRAPH *problem within an approximation factor of 2.*

PROOF. We prove the lemma by modifying the analysis of Charikar [3]. Let $f(S) = \frac{E(S)}{g(S)}$ be the objective function to optimize, over a subset S of the original set of nodes V. We first produce an upper bound on the optimal solution. Consider any *assignment* of each edge $e = (u, v)$ in the graph to either node u or node v. Let $d_{\text{in}}(u)$ be the *number* of edges assigned to node u, and let $D = \max_u\{\frac{d_{\text{in}}(u)}{g(u)}\}$; recall that $g(u)$ is the node weighting function.

Consider the optimal solution S^*. Each edge in $E(S^*)$ must be assigned to a node in S^*. Thus, we have

$$|E(S^*)| = \sum_{u \in S^*} d_{\text{in}}(u) \leq \sum_{u \in S^*} Dg(u) = Dg(S),$$

from which it follows that

$$\max_{S \subseteq V}\{f(S)\} \leq D. \tag{2}$$

Now consider the specific assignment constructed during the execution of the greedy algorithm. Initially all edges are unassigned. When a node u with minimum weighted degree $\frac{d(u)}{g(u)}$ is deleted from S, all edges currently in S and incident to u are assigned to u. We maintain the assignment that all edges between nodes currently in S are unassigned, while all other edges are assigned.

Let D be defined as before, for this specific assignment constructed during the execution of the algorithm. Also let f_G be the maximum value of $f(S)$ for all sets S obtained during the execution of the algorithm. Consider a single iteration of the greedy algorithm, let S be the set of nodes currently alive, and let $u_{\min}$ be the node deleted at that iteration. Since $u_{\min}$ is selected for deletion it should hold

$$\frac{d_S(u_{\min})}{g(u_{\min})} \leq \frac{d_S(v)}{g(v)},$$

for all nodes $v \in S$, and where d_S is the degree of a node in the subgraph defined by S. From the previous inequality it follows that

$$\frac{d_S(u_{\min})}{g(u_{\min})} \leq \frac{\sum_{v \in S} d_S(v)}{\sum_{v \in S} g(v)} = 2\,\frac{|E(S)|}{g(S)} \leq 2\,f(S) \leq 2\,f_G.$$

Since edges are assigned to $u_{\min}$ only when $u_{\min}$ is deleted, we have $d_S(u_{\min}) = d_{\text{in}}(u_{\min})$, and considering the specific node $u^*_{\min}$ for which the maximum D is materialized, we have

$$D = \frac{d_{\text{in}}(u^*_{\min})}{g(u^*_{\min})} = \frac{d_S(u^*_{\min})}{g(u^*_{\min})} \leq 2\,f_G. \tag{3}$$

Combining Equations (2) and (3) proves that our modified greedy algorithm is a factor-2 approximation to the weighted version of the DENSESTSUBGRAPH problem. □

Subsequent greedy steps. The discussion so far has shown how to perform the first greedy step of the SETCOVER algorithm. Our algorithm, shown as Algorithm 1, iteratively applies the steps until all edges of E are covered. The output of the oracle for the DENSESTSUBGRAPH problem needs to consider the choices done in previous steps. This is why the `DensestSubgraph` function takes the sets H, L and Z as inputs, and uses them as follows.

The sets H and L are used to update the weights $g(v)$. If some previous step has added an edge $(x \to w)$ to the set H, then the cost of pushing over that edge has already been paid, and we update $g(x) = 0$ for all hub-graphs $G(w)$ for which $x \in X(w)$. Similarly, if an edge $(w \to y)$ is already in the set L, then we update $g(y) = 0$ for all hub-graphs $G(w)$ for which $y \in Y(w)$.

The set of edges covered by a hub-graph only includes elements of Z that have not been already covered. Therefore, the density function of the DENSESTSUBGRAPH oracle is defined as $d(S) = |E(S) \cap Z|/g(S)$.

Algorithm 1 CHITCHAT

```
Input: Directed graph $G = (V, E)$;
Output: Dissemination schedule $(H, L)$;
$Z \leftarrow E$;                                  {Uncovered edges}
$\mathcal{Q} \leftarrow \emptyset$;                {A priority queue}
$H \leftarrow \emptyset$;                          {Push edges}
$L \leftarrow \emptyset$;                          {Pull edges}
{Determine the first DENSESTSUBGRAPH oracle output}
for all $w \in V$ do
    Form maximal hub-graph $G(w)$;
    {Find densest subgraph $S$ in $G(w)$ with density $d(S)$}
    $(S, d(S)) = $ DensestSubgraph$(G(w), H, L, Z)$;
    {Insert subgraph $S$ in priority queue with cost $\frac{1}{d(S)}$}
    Insert$(\mathcal{Q}, S, \frac{1}{d(S)})$;
{Greedy steps for SETCOVER}
while $(|Z| > 0)$ do
    $S \leftarrow$ ExtractMin$(\mathcal{Q})$;     {Extract min-cost subgraph}
    $Z \leftarrow Z \setminus E(S)$;              {Edges $E(S)$ covered}
    {Add $S$ to the solution}
    $H \leftarrow H \cup \{\{S.X\} \to w\}$;
    $L \leftarrow L \cup \{w \to \{S.Y\}\}$;
    {Update the DENSESTSUBGRAPH oracle output}
    for all $G(w)$ that contain edges of $E(S)$ do
        Let $S^*$ be the current densest subgraph of $G(w)$;
        Remove$(\mathcal{Q}, S^*)$;
        $(S, d(S)) = $ DensestSubgraph$(G(w), H, L, Z)$;
        Insert$(\mathcal{Q}, S, \frac{1}{d(S)})$;
return $(H, L)$
```

Approximation guarantee. The solution of our algorithm has a logarithmic-factor approximation due to the greedy algorithm for SETCOVER. If we use an oracle for the DENSESTSUBGRAPH problem that provides the exact solution, no additional loss in quality incurs. Lemma 1 shows that if we use the greedy algorithm analyzed by Charikar [3] as an oracle for the DENSESTSUBGRAPH problem, the combined approximation factor is $\mathcal{O}(2 \cdot \ln n) = \mathcal{O}(\ln n)$. This leads to the following result.

Theorem 4 *The* DISSEMINATION *problem can be solved with an $\mathcal{O}(\ln n)$-factor approximation guarantee, using the mapping to* SETCOVER *problem, and applying the greedy algorithm with an oracle to the* DENSESTSUBGRAPH *problem.*

3.2 The PARALLELNOSY heuristic

We now introduce a greedy heuristic to solve the DISSEMINATION problem, which we call PARALLELNOSY. PARALLELNOSY improves the scalability of CHITCHAT by introducing two key simplifications. First, it only considers *predefined hub-graph structures*, thus eliminating the expensive step of finding the densest subgraph among all hub-graphs centered in a given hub node. Second, it can be run as a *parallel* algorithm, which takes multiple parallel optimization choices instead of selecting the globally best choice at each iteration; the algorithm uses *locking* to prevent making conflicting choices. Like CHITCHAT, PARALLELNOSY is designed to optimize a static social graph. Incremental updates can be handled as described in Section 3.3.

Overview. PARALLELNOSY proceeds in iterations; an overview of an iteration is shown in Algorithm 2. Eventually, the cost converges to some local minimum, so executing further iterations does not improve the cost any longer and the algorithm terminates. The algorithm uses three sets, initially empty: the push edges H, the pull edges L, and the edges C covered by some hub.

Every iteration proceeds in three phases: *candidate selection*, *edge locking*, and *scheduling decision*.

Algorithm 2 PARALLELNOSY: overview of one iteration

```
Input: Directed graph $G = (V, E)$;
Input: Current dissemination schedule $(H, L)$;
Input: Set $C$ of edges covered through some hub;
Output: Updated dissemination schedule $(H, L)$;
Output: Updated set $C$ of edges covered through some hub;
    { Phase 1: Candidate selection, parallel for each edge $w \to y$}
for all $w \to y \in E$ s.t. $w \to y \notin C$ do
    $X \leftarrow \{x \mid (x \to w) \in E \setminus C \wedge (x \to y) \in (E \setminus (C \cup H \cup L))\}$;
    if $s(X, w, y) - c(X, w, y) > 0$ then
        $G(X, w, y)$ is a candidate hub-graph;
        for all $u \to v \in G(X, w, y)$ do
            lock $u \to v$ with priority $s(X, w, y) - c(X, w, y)$;
    { Phase 2: Edge locking, parallel for each edge $u \to v$}
for all $u \to v \in E$ do
    collect all lock requests for $u \to v$;
    grant edge lock to the hub-graph with highest priority;
    { Phase 3: Scheduling decision, parallel for each hub-graph }
for all hub-graphs $G(X, w, y)$ do
    if $G(X, w, y)$ is candidate and has all locks granted then
        add $w \to y$ into $L$;
        for all $x \in X$ do
            add $x \to w$ into $H$;
            add $x \to y$ into $C$;
    else
        $X' \leftarrow$ subset of $x' \in X$ s.t. $G(X, w, y)$ was granted locks for
            $x' \to y$ and $x' \to w$ ;
        if $s(X', w, y) - c(X', w, y) > 0$ then
            add $w \to y$ into $L$;
            for all $x' \in X'$ do
                add $x' \to w$ into $H$;
                add $x' \to y$ into $C$;
merge all updates to $H$, $L$ and $C$
return $(H, L, C)$;
```

The *candidate selection* phase chooses hub-graphs based on the observation that, in social networking systems, production rates are often smaller than consumption rates, so pull edges are more expensive than push edges. In terms of the hub-graph of Figure 3, candidate selection looks for hub-graphs where the set Y consists of a single node y, covering many $x \to y$ edges with multiple (cheap) $x \to w$ push edges and only one (expensive) $w \to y$ pull edge. One such hub-graph is considered a *candidate* only if selecting it reduces cost compared to the hybrid schedule of Silberstein et al. [11]. The algorithm can stop if no such candidates are found.

Candidate selection generates candidate hub-graphs in parallel; therefore, some candidates may require to modify the schedule of shared edges in an inconsistent, wasteful manner. The *edge locking* phase prevents such conflicts: if multiple hub-graphs try to modify the schedule of an edge, the one leading to the highest cost reduction obtains the lock to change it.

In the *scheduling decision* phase, each hub-graph changes the schedule of the edges it got a lock for. Given the structure of social graphs, only few candidate hub-graphs achieve to acquire locks for all their edges. An edge, in fact, could be shared by a very large number of hub-graphs. In order to perform more optimizations at each iteration, a candidate hub-graph that gets locks only for a subset of its edges reevaluates if it can achieve gains using only its locked edges. This suffices to prevent conflicts while achieving faster convergence.

We now discuss the three phases in detail.

Phase 1: Candidate selection. The first phase examines available hub-graphs and evaluates the cost reduction they can give, compared to the current solution. For each edge $w \to y$, candidate selection builds a hub-graph similar to the one of Figure 3 where $Y = \{y\}$. Each hub-graph is built and evaluated independently of each other. Therefore, candidate selection can be executed in parallel by multiple processes, each responsible for one hub-graph.

For each hub-graph $G(X, w, y)$, the set X is built by selecting common predecessors of w and y, with two conditions. The first condition is that the edge $x \to w$ is not covered already through a hub; since the hub requires pushing over this edge, we do not want to "undo" optimizations done in previous iterations that covered the edge $x \to w$ through some other hub. The second condition is that the cross-edge $x \to y$ is not covered already through some hub, and that it has not been scheduled to be a push or pull; in these cases, in fact, covering the edge $x \to y$ through w would be useless. The two conditions can be formally expressed by adding nodes x in X such that $x \to w \notin C$ and $x \to y \notin (C \cup H \cup L)$. For similar reasons, we require that $w \to y \notin C$.

Candidate hub-graphs must cover new edges with a lower cost than the hybrid schedule of Silberstein et al. [11], which covers each edge $x \to y$ with a cost $c_*(x \to y) = \min\{r_p(x), r_c(y)\}$. Selecting a hub-graph $G(X, w, y)$ saves the cost of covering cross-edges between nodes in X and y, resulting in *saved cost*

$$s(X, w, y) = \sum_{x \in X, (x \to y) \in E} c_*(x \to y).$$

The *positive cost* of a hub-graph $G(X, w, y)$ is computed by considering the edges that need to be scheduled as push or pull edges, respectively. The positive cost on an edge $e = x \to w$ is

$$c_X(e) = \begin{cases} r_p(x) & \text{if } e \in L \setminus H \\ r_p(x) - c_*(e) & \text{if } e \notin (H \cup L) \\ 0 & \text{if } e \in H \end{cases}$$

In the first case, if the edge e is in $L \setminus H$, PARALLELNOSY has previously decided that the edge is served by a pull, but not by a push. Selecting G mandates that the edge must be served by a push too, hence incurring an additional cost of $r_p(x)$. In the second case, if the edge e is not in $H \cup L$, PARALLELNOSY has not scheduled the edge yet. The additional cost of pushing over e depends on $r_p(x)$ and the cost $c_*(e)$ of covering e with the hybrid schedule. Finally, if e is already served by a push, there is no additional cost. The cost $c(w \to y)$ of the edge $w \to y$ is specular.

The overall positive cost of the hub-graph is thus

$$c(X, w, y) = \sum_{x \in X} c_X(x \to w) + c(w \to y).$$

The PARALLELNOSY heuristic considers a hub-graph $G(X, w, y)$ as a *candidate* if its saved cost is higher than its positive cost.

Phase 2: Edge locking. Before selecting candidate hub-graphs for scheduling, PARALLELNOSY needs to make sure that the speculative cost reductions calculated during candidate selection are indeed correct. In fact, candidate selection of each hub-graph assumes that no other hub-graph will be selected in parallel. PARALLELNOSY uses *locking* to select hub-graphs in parallel while preserving the correctness of independent cost estimations.

In the edge locking phase, each candidate hub-graph tries to *lock* its edges. Edge locks are assigned in parallel: there is one separate process responsible for evaluating lock requests for each edge $u \to v$ in the graph. The edge locking process responsible for $u \to v$ receives the gain value $s(X, w, y) - c(X, w, y)$ for each candidate hub-graph that includes $u \to v$. The process assigns the lock only to the hub-graph with highest gain.

Phase 3: Scheduling decision. During the last phase of PARALLELNOSY, one process is responsible for handling each candidate hub-graph $G(X,w,y)$. For each edge $u \to v$ in $G(X,w,y)$, the process receives information on whether $G(X,w,y)$ has successfully locked $u \to v$. If $G(X,w,y)$ receives locks for all its edges, the process *selects* the hub-graph for the schedule, that is, it adds all edges $x \to w$ with $x \in X$ into H, all edges $x \to y$ with $x \in X$ into C, and the edge $w \to y$ into L. Locking ensures that there are no conflicts while modifying the sets H, L and C in parallel: each edge will be added to only one of these sets by only one process. The final value of H, L and C at the end of the iteration is the union of all sets determined during the scheduling decision phase.

If a candidate hub-graph $G(X,w,y)$ only receives locks for a strict subset of edges, the process builds a hub-graph $G'(X',w,y)$ using only the locks it got. The set $X' \subset X$ includes only the nodes x' such that both edges $x' \to w$ and $x' \to y$ were successfully locked. The process applies the scheduling changes induced by $G'(X',w,y)$ if $s(X',w,y) - c(X',w,y) > 0$, where the costs are determined as in the candidate selection phase. Locking still guarantees the absence of conflicts.

Implementing PARALLELNOSY **with MapReduce.** The PARALLELNOSY algorithm is designed to be parallel, so it can be easily implemented using MapReduce [6]. This implementation is the one we used to evaluate the approach. We now describe in more detail the issues pertaining to the MapReduce implementation; we assume that the reader is familiar with the MapReduce architecture.

Prior to the first iteration of PARALLELNOSY the implementation executes a preliminary job that builds a hub-graph $G(X,w,y)$ for each edge $w \to y$. In particular, each hub-graph detects the cross-edges that it could potentially cover. Cross-edges detection is expensive since it is requires fetching edges at distance two from the hub node w. In very large social graphs, workers responsible for high-degree hub nodes may consume a large amount of memory to detect cross-edges, potentially leading to job failures. We overcome this problem by fixing an upper bound b on the number of detected cross-edges. A worker responsible for a large hub-graph starts by loading in memory the first b edges it received. If some loaded edge is not found to be part of the hub-graph, it is replaced with the next edge that has not been yet loaded.

Phase 1 is executed by the *map phase* of MapReduce, where each mapper takes a hub-graph $G(X,w,y)$ as input. If the hub-graph is a candidate then the mapper requests to lock all edges of the graph by outputting a key-value pair for each edge $u \to v$ in $G(X,w,y)$. The key is the id of the edge $u \to v$; the value contains the id of the edge $w \to y$, which uniquely denotes the hub-graph $G(X,w,y)$, together with its gain $s(X,w,y) - c(X,w,y) > 0$.

Phase 2 is executed by the *reduce phase* of MapReduce, where each reducer receives all lock requests for a given edge $u \to v$. The reducer assigns the lock to the hub-graph with highest gain. The output is a key-value pair where the key is the id of the edge $w \to y$ of the hub-graph that got the lock and the value is the id of the locked edge $u \to v$.

Phase 3 is implemented as a reduce-only job, in which each hub-graph receives the list of edge locks it was granted, and outputs a list of edge updates that represent its scheduling decisions.

After Phase 3, an additional MapReduce job merges all scheduling decisions and disseminates these change to the inputs of the next iteration. Every update to an edge $u \to v$ needs to be sent not only to the hub-graphs centered in u and v, but also to the hub-graphs centered in neighbors of u and v, since these could have $u \to v$ as a cross-edge.

Using a push approach for the final update dissemination is simpler but results in a flood of information that makes the execution of one iteration much slower. Therefore, our implementation uses a pull approach and two MapReduce jobs: in the first job, hub-graphs having $u \to v$ as cross-edge send a notification to the hub-graphs centered in u and v saying that they are interested in updates to $u \to v$. Updates for the edge are propagated only if they are indeed available. This reduces the load on the network and significantly speeds up the execution time of an iteration.

3.3 Incremental updates

PARALLELNOSY and CHITCHAT optimize a static social graph. Incremental updates to the graph can be trivially implemented as follows: if an edge is added, it is served directly, choosing the cheaper between a push and a pull policy. If a pull edge $u \to v$ is removed, where u is a hub, then all edges pointing to v that are covered via u are served directly. The case where v is a hub and the edge is served by a push is similar. Over time, graph updates let the quality of the dissemination schedule degrade, so our algorithms can be executed periodically to re-optimize cost. The experimental evaluation of Section 4 indicates that our algorithm does not need to be re-executed frequently.

4. EVALUATION

In this section, we evaluate the throughput performance of the proposed algorithm, contrasting it against the best available scheduling algorithm, the hybrid policy of Silberstein et al. [11].

Our evaluation is both analytical, considering our cost metric of Section 2.1, and experimental, using measurements on a social networking system prototype. We show that the PARALLELNOSY heuristic scales to real-world social graphs and doubles the throughput of social networking systems compared to hybrid schedules. On a real prototype, PARALLELNOSY provides similar throughput as hybrid schedules when the system is composed by few servers; as the system grows, the throughput improvement becomes more evident, approaching the 2-factor analytical improvement.

We also evaluate the relative performance of the two proposed algorithms PARALLELNOSY and CHITCHAT. This comparison is relevant because PARALLELNOSY is more scalable while CHITCHAT is theoretically superior.

4.1 Input data

We obtain datasets from two social graphs: `flickr`, as of April 2008, and `twitter`, as of August 2009. The `twitter` graph has been made available by Cha et al. [2]. `flickr` has 2 409 730 nodes and 71 345 981 edges; `twitter` has 82 949 778 nodes and 1 423 194 279 edges.

Our algorithms also require input workloads: production and consumption rates for all the nodes in the network. As we do not have access to real workloads for neither of the two datasets, we synthetically generate workloads using observations from the literature. It has been observed by Huberman et al. that nodes with many followers tend to have a higher production rate, and nodes following many other nodes tend to have a higher consumption rate [8]. To model this behavior, we set the production and consumption rates of the nodes to be proportional to the logarithm of their in- and out-degrees, respectively. We consider a reference ratio of average production rate vs. average consumption rate equal to 5, as observed by Silberstein et al. [11].

4.2 Social piggybacking on large social graphs

We run our MapReduce implementation of the PARALLELNOSY heuristic on the full `twitter` and `flickr` graphs. We use 1500 cores of a shared Hadoop cluster. Executing the first iteration on the larger `twitter` graph takes about 1 hour; the execution time for subsequent iterations decreases to about 45 minutes from the fourth iteration on, as fewer optimization opportunities are left.

As discussed in Section 3.2, very large social graphs may contain millions of cross-edges for a single hub-graph. This is the case of the `twitter` dataset, so we execute the cross-edges detection phase at every cycle of ParallelNosy, with an upper bound of 100,000 cross-edges per hub-graph. We execute cross-edges detection only once for `flickr`, as the graph is significantly smaller.

For the `twitter` graph, the amount of memory used by individual MapReduce workers exceeds in some cases the RAM capacity allocated to these workers, which is 1GB. Such cases occur because the graph is so densely connected that building full hub-graphs is sometimes unfeasible. We solve this problem with a simple approach: given a hub-graph for an edge $w \to y$, if the two-hop neighborhood of the hub w is too large, we remove some nodes from the predecessor set of w, and in particular the predecessors that have no cross edges to y and that will never be included in a hub-graph $G(X, w, y)$. With this conservative modification we still cover all edges of the original graph; we only make the computation feasible at the cost of missing some optimization opportunities.

Predicted throughput. We quantify the performance of our algorithms by measuring their throughput compared against a baseline. Consider the request schedule (H, L) produced by an algorithm A for a given input, and assume that it achieves cost c_A (see Section 2.1) for that input. We define the *predicted throughput* t_{A} of algorithm A to be the inverse of the cost, i.e., $t_{\mathrm{A}} = c_{\mathrm{A}}^{-1}$. We use the term *predicted* to emphasize that this throughput estimate is based on our cost function, as contrasted to the actual throughput reported in the next section, which is based on measurements obtained with our prototype implementation.

We use as baseline the hybrid schedule of Silberstein et al. [11], which is the best available algorithm. We refer to this baseline as FeedingFrenzy algorithm, or simply as FF. Hybrid schedules are per-edge optimizations which can be easily calculated by visiting each edge of the social graph once.

To compare with FF, we define the *predicted improvement ratio* of an algorithm A as $t_{\mathrm{A}}/t_{\mathrm{FF}}$, where t_{A} is the predicted throughput of the algorithm A and t_{FF} is the predicted throughput of the baseline. Algorithm A can be either ParallelNosy or ChitChat. A relative throughput greater than 1 indicates that the algorithm A outperforms the FF baseline.

Figure 4 shows the predicted improvement ratio of ParallelNosy for full social graphs over the FF baseline. Running more iterations of ParallelNosy leads to higher throughput improvement. For both social graphs, the throughput of the ParallelNosy schedule increases sharply during the first iterations and it quickly stabilizes. The larger stabilization time for `twitter` is due to the incremental detection of cross-edges at every cycle, as discussed before.

The throughput increase of ParallelNosy, a factor of about 2 for both datasets, is substantial. The `twitter` graph enables higher throughput performance since it is denser than `flickr`.

Incremental updates. ParallelNosy addresses the problem of optimizing a static social graph, but we also described a simple approach for incremental updates. In the experiment illustrated by Figure 5 we investigate the effect of executing ParallelNosy after a batch of k edges is added to the graph. We start by running ParallelNosy on the half of the edges of `flickr`, selected at random. We then add k randomly selected edges and optimize the graph using two different policies: an incremental policy, which uses the baseline for the last k edges, and a static policy, which re-optimizes the graph again using ParallelNosy after adding the last edges. Figure 5 shows that incremental policy is more expensive, but it degrades slowly compared to the static one; we magnify the y axis to better show the degradation. If the heuristic is applied once every 10^7 added edges, which is almost one third of the initial graph, the throughput increase remains stable. Therefore, after executing an initial optimization of the social graph, a large number of edges can be added before a re-optimization becomes needed.

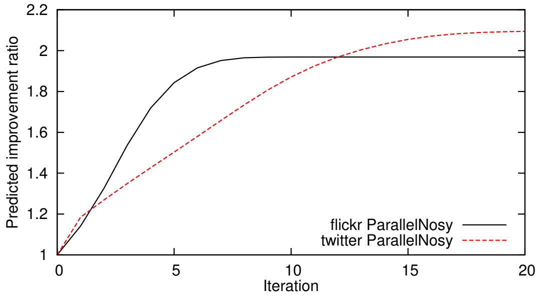


Figure 4: Predicted improvement ratio of ParallelNosy.

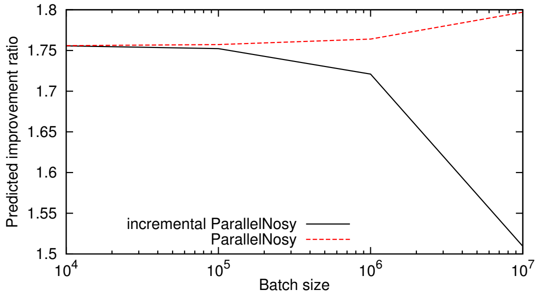


Figure 5: Predicted improvement ratio of static and incremental ParallelNosy, starting from half flickr graph and adding increasingly large batches of new edges.

4.3 Prototype performance

In the previous section we evaluated our algorithms in terms of the predicted cost function that the algorithms optimize. In order to obtain a more realistic performance evaluation, we test the proposed algorithms on a real social networking system prototype and we measure actual throughput. Our results show that ParallelNosy increases the throughput of our social networking prototype. We start by describing our system.

Description of the prototype. The architecture of our prototype is the one shown in Figure 1. We consider an event-stream index, where user views contain references to events. In such a system, serving event-stream queries entails two steps; the first step is assembling the event stream, which involves querying user views over the social graph; the second step is event-stream rendering, which involves retrieving the text of the event, comments, pictures, expanding links etc. Our implementation focuses on the first step of assembling the event-stream, which queries user views over a social graph. Updates insert events as (*user_id*, *event_id*, *timestamp*) tuples into user views; queries return the 10 latest events across all friends. The tuple size is 24 bytes.

Our prototype uses Java for the application logic and memcached as data store for the views; we added a thin layer on top of memcached, at the server side, to aggregate and filter out tuples in case of queries and to trim views when they contain too many events.

The pseudocode of application logic servers is illustrated in Algorithm 3. For simplicity, we do not show the logic for handling

Algorithm 3 Pseudocode of application servers

```
upon receive update $d$ from user $u$ do
    $h[u] \leftarrow$ get-push-set-from-schedule($u$);
    for all $s : \exists\, v \in h[u]$ stored by $s$ do
        send $d$ to data store $s$;
upon receive update ack from server $s$ do
    if received update acks from all data stores $s \in h[u]$ then
        send ack to the front-end server handling query from $u$;
upon receive query from user $u$ do
    $l[u] \leftarrow$ get-pull-set-from-schedule($u$);
    $r[u] \leftarrow \emptyset$;
    for all $s : \exists$ view $v \in l[u]$ stored by data store $s$ do
        send query to data store $s$;
upon receive new query reply $n$ from a data store do
    $r[u] \leftarrow$ filter($n, r[u]$);
    if received query replies from all data stores $s \in l[u]$ then
        send $r[u]$ to the front-end server handling query from $u$;
```

message losses and crashes, and for ensuring that each user has at most one outstanding request at any given time. Application logic servers execute the same operations regardless of the adopted schedule; schedules determine the push-sets $h[u]$ and pull-sets $l[u]$ used in update and query operations, respectively. Push and pull sets for all users are kept in memory. The *filter* operation is generic: in our example, it keeps the 10 latest events in $r[u]$. Reply lists $r[u]$ are kept in memory so the cost of filtering is negligible. We use *batching*: when processing a user query, application servers send at most one query per data store server s, which replies with a list of events filtered from all views $v \in l$ stored by s. Most data store layers offer a query/update client interface that, given a set of views, transparently communicates with servers using batching.

All our experiments are run on a large cluster of Intel Xeon servers with sixteen 2.4 GHz cores, 24 GB of main memory and a Gigabit network.

The workload for our evaluation consists of a sequence of user queries and updates received by the application-logic servers, which act as data-store clients; see Figure 1. In the following, we refer to application-logic servers as *clients*, as they are clients for the data store, and to data-store servers as *servers*. We consider the `flickr` graph, and generate a workload using the same parameters as in the previous section. For simplicity, clients keep the social graph and the related request schedule in main memory. They translate each query and update into one or more queries and updates to servers. Servers keep user views in main memory.

Data partitioning. We refer to *data partitioning* in social networking systems as the mapping from user views, or equivalently nodes of the social graph, to servers. Due to the use of batching in our prototype, data partitioning has an impact on actual throughput: for example, if two neighboring nodes u and v are mapped to the same server, disseminating events over the edge $u \rightarrow v$ has zero cost. Using data partitioning information as input of the DISSEMINATION problem is attractive, but has two main drawbacks. First, this information might be hidden as internal logic of the data store layer and might be unavailable. Second, data partitioning is highly dynamic and can be modified often during the lifetime of a system, for example, if servers fail or if new servers are added to the system. Including information on data partitioning as an input would make incremental updates more complex and frequent. Therefore, our definition of the DISSEMINATION problem does not take data partitioning information as input. Our evaluation prototype, however, does use data partitioning and batching, showing that this additional information is not essential to achieve significant performance gains. The prototype uses a simple partitioning approach that is common in practical data store layers: the view of a user u is stored in a random server, selected by hashing the id of the user.

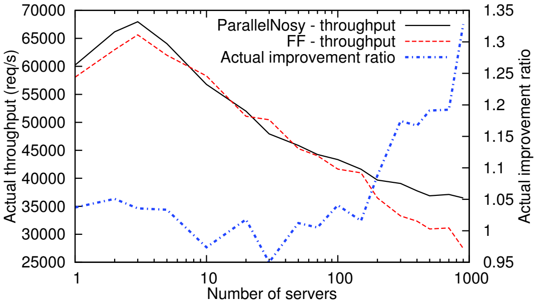


Figure 6: Actual per-client throughput of our prototype as a function of the number of servers. The first two lines have y axis on the left, the third on the right.

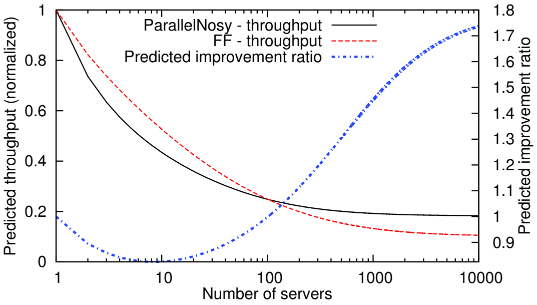


Figure 7: Predicted throughput as a function of the number of servers. The first two lines have y axis on the left, the third on the right.

Actual throughput. Our evaluation focuses on *actual throughput*, expressed as the number of requests completed per second in our prototype. For the measurements, we consider a request to be completed when the front-end processing it (see Figure 1) receives a reply. Since queries involve only simple processing of in-memory data structures, the latency per request is very low unless the system becomes saturated.

Since all clients are identical and operate independently from each other, we evaluate the throughput improvement per-client. We compare against the throughput obtained by the same prototype when the hybrid schedule of Silberstein et al. [11] is used to compute the push-sets $h[u]$ and the pull-sets $l[u]$ of Algorithm 3; we keep referring to this baseline as FF.

Figure 6 reports per-client throughput of our prototype. Clients have more load per request than servers: given a single request, clients may send multiple queries to servers, while each server only has to process at most one query. As we increase the number of servers in the system, clients are likely to contact more servers and send more queries per request; this reduces the absolute per-client throughput. However, a larger number of servers supports a larger number of clients, resulting in improved actual throughput. We found that, if the network does not become a bottleneck, the overall throughput using n clients and n servers is about n times the per-client throughput with n servers.

PARALLELNOSY is particularly effective and scalable to systems with billions of requests per second. According to our measurements, hundreds of servers are necessary to support this load. In systems with 200 or more servers, throughput benefits significantly from the use of PARALLELNOSY. Figure 6 shows that the throughput improvement is about 20% with 500 servers, and about 35% with 1000 servers. Random data partitioning sometimes makes the relative throughput curve irregular, especially when the system is small, but the trend is clear: the throughput gain of PARALLELNOSY increases when the system size grows.

With a lower number of servers, the two scheduling algorithms lead to similar cost, with the baseline sometimes performing slightly better. This is because with fewer nodes, there is a higher likelihood that, for any given edge $u \rightarrow v$, both u and v are mapped to the same server S. The cost of a push or pull over the edge in this case is just the cost of sending a request to S, which is needed anyway every time u updates or v queries. Since serving $u \rightarrow v$ comes for free, there is no need to prune it. Our algorithm, however, may try to prune this edge anyway by making u and v communicate through some hub node w. If w is mapped to a data store different than S, the algorithm may schedule an additional or expensive pull request. With a higher number of servers, however, it becomes less likely that u and v are mapped to the same server.

Figure 7 reports the *predicted* throughput of the request schedules. After obtaining the schedules, we calculate their predicted throughput (see Section 4.2), this time considering the effect of data placement: if two views are mapped to the same server, a single message can query both views at once. We normalize predicted throughput and divide it by the (optimal) predicted throughput obtained with only one server. The consistency between the experimental throughput results and our predicted cost evaluation is striking. The ratio between PARALLELNOSY and FF follows a very similar trend as in our evaluation. FF results in higher throughput in smaller systems, but PARALLELNOSY outperforms in systems with more than 200 servers. The actual values of the relative predicted and actual throughput match very well. Figure 7 considers even larger systems than Figure 6, with up to 10000 servers.

As the number of servers grows, the predicted throughput of Figure 7 converges to the results reported in Figure 4, where data placement is not considered. This is because as the number of servers increases, the likelihood of having neighboring nodes randomly placed in the same server decreases, and thus, the effect of data placement becomes negligible.

Beyond per-client throughput, a schedule supporting heavy workloads must balance load, which in our case is the query rate per server. Figure 8 compares the load balancing capabilities of PARALLELNOSY and FF schedules using this load metric. We plot average values; error bars represent the variance. Note that, since the y axis is logarithmic, the divergence between the algorithms and the error bars on the right side of the graph are magnified. As the number of servers grows, the average load per server decreases for both algorithms. Figure 8 shows that both algorithms produce well-balanced schedules, especially in larger systems.

4.4 The potential of social piggybacking

The previous experiments show that PARALLELNOSY is an effective heuristic for real-world large-scale social networking systems. However, we do not know how close PARALLELNOSY can get to an optimal social-piggybacking schedule. Thus, in this section we evaluate PARALLELNOSY against the CHITCHAT algorithm, which has provable approximation guarantees. Our objective is to demonstrate that the potential of social piggybacking to improve further the (already good) performance of PARALLELNOSY.

CHITCHAT is a relatively expensive centralized algorithm that does not scale to very large social graphs; this constraint restricts our evaluation to samples of the `twitter` and `flickr` social graphs that consist of 5 million edges.

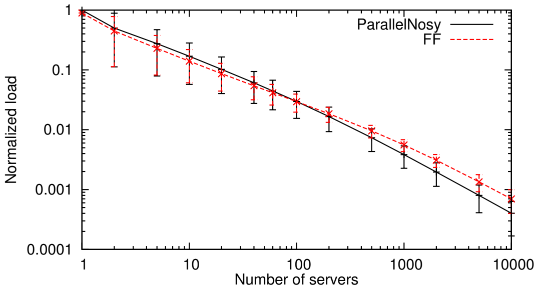


Figure 8: Load balancing – Query rate per server.

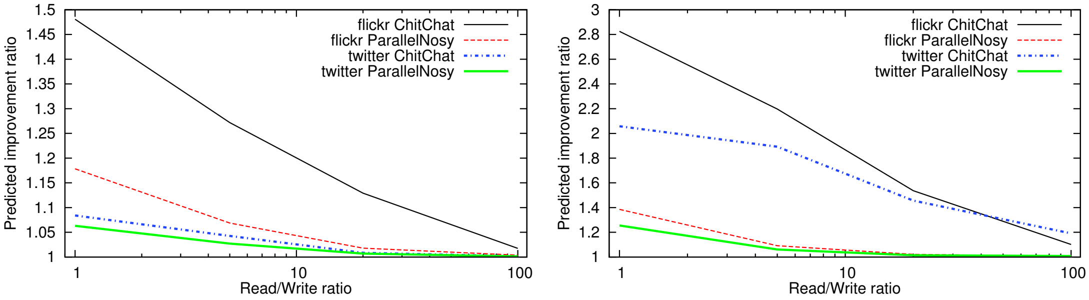
We aim at obtaining samples that resemble real-world graphs. The problem of sampling a graph in a way that the resulting subgraph maintains the properties of the original graph is an ongoing research problem. Therefore, we experiment with two different sampling methods: random-walk sampling and breadth-first sampling. In the experiments discussed below we use five graph samples; the plots report averages.

Figures 9 shows the predicted improvement ratio of CHITCHAT for random walk and breadth-first samples. The main result is that the difference between PARALLELNOSY and CHITCHAT is large, which points to an opportunity for new heuristics and further improvement with social piggybacking. Overall, by comparing these results with the ones shown in Figure 4 we see that the cost of PARALLELNOSY is lower in real social graphs than in the sampled graphs; consequently we expect that the cost of CHITCHAT in a real social graph would be substantially lower too. The results in the figures also confirm our observation that the graph-sampling technique impacts the performance of social-piggybacking.

The algorithms are more efficient on samples obtained by the breadth-first method than on samples obtained by the random-walk method. This difference is due to the positive correlation between the effectiveness of our schedules and the presence of hub nodes with high degree. In breadth-first sample graphs, the first sampled nodes have the same degree as in the original social graph. As for random-walk sampling, existing work has pointed out that it preserves certain clustering metrics; more precisely, in both the original and sampled graphs, nodes with the same degree have similar ratio of actual and potential edges between their neighbors [9]. However, other properties of the original graph may not be preserved; for example, edges of high-degree nodes may be pruned out. This reduces the relative gain of social piggybacking since the hybrid schedule of Silberstein et al. (our baseline) uses per-edge optimizations that do not depend on the degree of nodes.

The plots show the performance of the algorithms as a function of the read/write ratio, that is, the ratio between the average consumption and production rates. We set this ratio as high as 100, which is 20 times the reference value, to represent the extreme case of a workload heavily dominated by reads. Intuitively, if users consume information every second while producing information only once a day then the hybrid schedule, which uses push edges to spread the (rare) events through the network, should be nearly-optimal. The experiments confirm this intuition.

To conclude, the results of this section reflect that the potential of social piggybacking go beyond the performance of PARALLELNOSY, and suggest interesting future work on the design of techniques to scale the CHITCHAT algorithm to very large datasets.



(a) Random-walk sampling.

(b) Breadth-first sampling.

Figure 9: Performance comparison of CHITCHAT **and** PARALLELNOSY **on social graph samples.**

5. RELATED WORK

Similar to the MIN-COST problem of Silberstein et al. [11], our DISSEMINATION problem takes as input the consumption and production rates of users, together with the social network, and uses these rates in the definition of the cost function. We generalize MIN-COST as a graph-propagation problem, which encompasses multiple practical propagation policies. This enables taking advantage of the high clustering coefficient of social graphs and leads to substantial gains, as shown by our evaluation.

Pujol et al. describe SPAR, a new storage layer for social networking systems. When a user u produces a new event, SPAR first stores it in its "master replica". This master replica is located together with "slave replicas" of all friends of u; logically, all these replicas form what we call the "view" of u. SPAR pushes new events of u asynchronously from the master replica of u (i.e., from the view of u) to all its slave replicas (i.e., to the views of all friends of u). Users contact only their own views for queries. In terms of throughput cost, SPAR uses an (asynchronous) *push-all* schedule (see Section 1), which, as shown in [11], is never more efficient than the hybrid schedule we used as our baseline. Note that all the schedules considered in this paper can be executed asynchronously; this can be modeled as discussed in Section 2.2.

The SPAR middleware enhances the data store layer with several complex functionalities for data partitioning, movement, and replication. By contrast, schedules produced by PARALLELNOSY can be used at the client-side of standard passive data stores, such as memcached or MySQL, so they do not require using a novel storage layer or middleware.

Our problem definition has some similarities with the work on optimal overlays in publish-subscribe systems initiated by Chockler et at. [4]. They compute an optimal graph of physical servers that minimizes edge degree. In our case, the social graph is given, the mapping of users to physical servers is not known, we minimize cost based on scheduling decisions and production and consumption rates, and we consider the additional bounded staleness constraint. Both problem definitions avoid the generation of useless messages by requiring that events are only sent to vertices that subscribe to the topic; in our case, only views of users that follow the producer of an event store the event.

6. CONCLUSION

Assembling and delivering event streams is a major feature of social networking systems and imposes a heavy load on back-end data stores. We have introduced social piggybacking, a promising approach to increase the throughput of event stream handling by identifying better request schedules.

We proposed two algorithms to compute request schedules that leverage social piggybacking. The CHITCHAT algorithm is an approximation algorithm that uses a novel combination of the SETCOVER and DENSESTSUBGRAPH and has an approximation factor of $\mathcal{O}(\ln n)$. The PARALLELNOSY heuristic is a parallel algorithm that can scale to large social graphs.

We used PARALLELNOSY to compute request schedules for the full Twitter and Flickr graphs. In small systems, we obtained similar throughput as existing hybrid approaches, but as the size of the system grows beyond a few hundreds of servers, the throughput grows significantly, reaching a limit of a 2-factor improvement. Evaluation on CHITCHAT shows that request schedules using social piggybacking have an even higher potential for cost reduction.

7. REFERENCES

[1] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34(2):203–221, 2000.

[2] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proc. of ICWM*, volume 14, page 8, 2010.

[3] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Proc. of APPROX*, pages 139–152, 2000.

[4] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub-sub with many topics: Problems, algorithms, and evaluation. In *Proc. of PODC*, pages 109–118, 2007.

[5] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

[8] B. A. Huberman, D. M. Romero, and F. Wu. Social networks that matter: Twitter under the microscope. *First Monday*, 14(1-5), 2009.

[9] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *Proc. of KDD*, pages 631–636, 2006.

[10] M. E. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[11] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *Proc. of SIGMOD*, pages 831–842, 2010.